REPORT NO. UIUCDCS-R-87-1388

The Design of a Multiprocessor Operating System

by Roy Campbell, Vincent Russo, Gary Johnston

December 1987

Department of Computer Science
1304 W. Springfield
Urbana, Illinois 61801

This work was supported in part by NASA NSG 1471 and by AT&T MetroNet.

TABLE OF CONTENTS

The Design of a Multiprocessor Operating System[1]

Roy Campbell, Vincent Russo, Gary Johnston

Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Ave., Urbana, IL 61801-2987

ABSTRACT

Evolving applications and hardware are creating new requirements for operating systems. Real-time systems, parallel processing, and new programming paradigms require large adaptive maintenance efforts to modernize existing operating systems. An alternative to such adaptive maintenance is to seek new operating system designs that exploit modern software engineering techniques and methodologies to build appropriately structured modular software. This paper describes an approach to constructing operating systems based on a class hierarchy and object-oriented design, and discusses the benefits and difficulties of realizing this design using C++.

1. Introduction

Choices is designed as an object-oriented system that supports user applications with an object-oriented operating system interface. The architecture embodies the notion of a *customized* operating system that is tailored to particular hardware configurations and to particular applications. Within one large computing system containing many processors, many different specialized operating systems may be integrated to form a general purpose computing environment. Choices is currently implemented on an Encore Multimax.[*]

Choices, a *Class Hierarchical Open Interface for Custom Embedded Systems*, provides a foundation upon which to construct sophisticated scientific and experimental software. Unlike more conventional operating systems, Choices is intended to exploit very large multiprocessors interconnected by shared memory and/or high-speed networks. Uses include applications where high-performance is essential like data reduction or real-time control. It provides a set of software classes that may be used to build specialized software components for particular applications. Choices uses a class hierarchy and inheritance to represent the notion of a family of operating systems and to allow the proper abstraction for deriving and building new instances of a Choices system. At the basis of the class hierarchy are multiprocessing and communication objects that unite diverse specialized instances of the operating system in particular computing environments.

The operating system was developed as a result of studying the problems of building adaptive real-time embedded operating systems for the scientific missions of NASA. Major design objectives are to facilitate the construction of specialized computer systems, to allow the study of advanced operating system features, and to support parallelism on shared memory and

[1] This work was supported in part by NASA under grant no. NSG1471 and by AT&T METRONET.

[*] Multimax is a trademark of Encore Computer Corporation.

networked multiprocessor machines. Example specialized computer systems include support for reconfigurable systems, robotics applications, network controllers, aerospace applications, high-performance numerical computations, and parallel language processor servers for IFP[14], Prolog, and Smalltalk. Examples of advanced operating system features include fault-tolerance in asynchronous systems, real-time fault-tolerant features, load balancing and coordination of very large numbers of processes, atomic transactions, and protection. Example hardware architectures include shared memory multiprocessors like the Encore Multimax and networked computers like the Intel Hypercube.

Choices was designed to address the following specific issues: the software architecture for parallel operating systems; the achievement of high-performance and real-time operation; the simplification and improved performance of interprocess communications; the isolation of mechanisms from one another and the separation of mechanisms from policy decisions.

Of particular concern during the development of the system was whether the class hierarchical approach would support the construction of entire operating systems. C++ was chosen because it supported classes while imposing negligible performance overhead at run-time. In particular, we decided to construct all parallel and synchronization features using C++ classes rather than by introducing new language primitives. Thus Choices is also a study of the adequacy of class hierarchies to abstract and support parallelism, synchronization, resource allocation, and other operating system concepts and to allow specializations of classes that facilitate efficient support for applications.

Fortunately for the designers of Choices, there has been a lot of operating system development that is directly applicable to our goals [4]. Abstracting the ideas from many different systems and reorganizing them into an object-oriented system has been a major concern of our design team.

In brief, Choices has been influenced considerably by

- the systems environment provided by UNIX,[*]
- the problem of allowing multiple processors to execute the kernel of an operating system,
- the reduction of process context switching overheads to better support real-time and high-performance applications,
- the open architecture of CEDAR [17],
- the use of classes to extend an efficient systems programming language rather than adopt a more complex programming language that has rich system programming features like Mesa or Ada,
- the exploitation of virtual memory to support object-oriented paradigms and communication primitives [1],
- the avoidance of designing another a distributed operating system but instead concentrating on the design of a parallel operating system,[2]
- the avoidance of the undesirable side effects of cacheing and cache flushing overhead in

[*] UNIX is a Registered Trademark of AT&T.

[2] Many distributed/multiprocessor UNIXs (UNIX United [3], LOCUS [11], Mach [1], RFS [13], RIDE [10], NFS [19], Encore Multimax UNIX (UMAX[*]) [8], Sequent Balance[*] 8000 UNIX [15]) still impose UNIX limitations on the parallelism and performance of applications.

cache-oriented multiprocessors,

- the avoidance of the inclusion of specific communication schemes into the lowest structures of an operating system kernel that restrict the possibilities of specializing kernel features to take advantage of communication patterns of the application or communication mechanisms of the hardware,[3]
- the avoidance of overhead from copying messages into and out of virtual memory (some cached systems may pay a double overhead),
- the support of real-time interrupts and global multiprocessor interrupts,
- the inclusion of parallel programming primitives (for example, the parallel creation of parallel processes),
- the provision of appropriate error recovery in parallel processing systems,
- the Clouds [2] notion of a user process accessing a user object,
- the provision of the smallest operating system that will support a particular application on a particular hardware,[4]
- the support for embedded, real-time, and server computing services that are provided by large numbers of fast processors connected together by shared memory and/or by a fast network,
- the support for a computational facility that is multitasked (it supports several concurrent applications), where each task may use multiple processors,
- the support for processes in an application that may have a high degree of communication,

In the subsequent sections, we discuss the class hierarchical organization of Choices, the various classes we have built to implement virtual memory, the concept of process, the notion of a persistent object and exception handling, performance of an object-oriented operating system, and the implementation of the system using C++.

2. The Choices Class Hierarchy Model

Several problems emerge when designing an extensible *family* of operating systems where each member can be specialized or customized for a particular application or hardware configuration. Each module within the system may have many different versions tailored for each different member of the family of operating systems. However, since the different versions of a module for different machines or applications all perform a similar function, large portions of different versions of a module will be identical. Customizing an operating system for a new application requires access to particular aspects of the code that may reside in many different

---

• UMAX is a trademark of Encore Computer Corporation.

• Balance is a trademark of Sequent Computer Systems.

[3] Message-oriented kernels like the V System kernel [7], Accent [12], Amoeba [18], and MICROS [20] build specific communication schemes into the lowest kernel structures. For example, some systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easy to adapt to support other approaches like "send process on read" or "remote procedure call on execute."

[4] General purpose operating systems often employ delayed bindings within their architectures to provide flexibility. Examples include communication schemes, file systems and additional kernel code to handle different architectures and configurations. Where several applications need to coexist within the same computing system, Choices allows these applications to each run on their own custom-built Choices operating system. Any communication required between the applications is supported by common Choices primitives and shared persistent objects.

modules.

A class hierarchy provides a solution to these problems. Particular instances of classes in the hierarchy are chosen and combined to produce a customized operating system for a specific architecture and application. Class inheritance provides for code reuse and enforcement of common interfaces. Customization of the operating system for new applications is guided and aided by the structure induced upon the system by the class hierarchy.

A class hierarchy gives more than ease of customization. It also gives us a conceptual view of how portions of an operating system interrelate. It is easier to understand and more flexible than traditional layered approaches to operating system design. A class hierarchy allows conceptual "chunking" of knowledge about portions of a system by learning the function of parent classes and inferring functionality about subclasses. Traditional layered operating system design approaches group large sections of functionality into a layer, but the interrelations of the layers are often complex and poorly understood. Also, changing a piece of a layer is in no way facilitated by the layering itself. However, in a well-designed Class Hierarchical model only the top few classes would need to be mastered to achieve a good overall view of the system. Class derivation gives a method to change specific parts without adversely effecting the whole structure.

Most of the major components of the class hierarchy for Choices are shown in Tables 1 through 4. Table 1 shows the major classes that comprise the first level in the hierarchy. *Object* is specialized into seven subclasses including *MemoryRange* and *SpaceList*. Each subclass redefines and adds new methods[5] to the methods defined for Object.

First Level Hierarchy of Classes		
Classes	Methods	
Object	**constructor**	**destructor**
↓ MemoryRange	*constructor*	*destructor*
↓ Process	*constructor*	*destructor*
↓ ProcessContainer	*constructor*	*destructor*
↓ Exception	*constructor*	*destructor*
↓ Filler	*constructor*	*destructor*
↓ SpaceList	*constructor*	*destructor*

Key to Hierarchy Tables	
Symbol	Meaning
Method	Definition of Method
↓	Subclass or Inherited Method
Method	Overloading of Method
—	Undefined Method

Table 1: Major Classes at First Level of Hierarchy

[5] In this paper the terms "member function" and "method" are used interchangeably.

Several guidelines have been used to develop an appropriate class hierarchy for Choices. All machine dependencies, operating system mechanisms (for example, page table management), operating system policy decisions (for example, schedulers) and design decisions are encapsulated within objects. In general, a design decision is often represented as a class with subclasses that represent the mechanisms that implement the decision. Table 3 shows an example in which the design separates *ProcessContainers* into containers that hold one thread and containers that hold many. The use of an instance of a subclass of a class as a representative instance of the class is often used to program specific policy decisions. For example, a FIFO scheduling discipline may be imposed on processes being added to a ProcessContainer by using an instance of the *FIFOScheduler* class to represent the ProcessContainer doing scheduling of processes for the system. A priority scheduled system can be created by replacing this ProcessContainer with an instance of a subclass that imposes the priority scheduling discipline on the processes it contains. Wherever possible, the class hierarchy is constructed so that similar sub-hierarchies can be specialized from a common ancestor hierarchy. Thus, for example, there is a hierarchy of classes representing memory management mechanisms shown in Table 2 that is specialized into virtual memory, real memory, and disk storage.[6] Overall, design progressed from class hierarchies that had a large fan out to hierarchies that had a small (2–7) fan out, and greater depth. In the future, multiple inheritance may further refine the hierarchy.

C++ was used to program all parts of the system. The language is efficient and portable. It implements object oriented programming semantics with minimal runtime overhead and thus is ideal for operating system programming. It is also easy to interface C++ to assembler in order to achieve things impossible in the language itself (for example, loading stack pointers and memory management unit registers.) The operating system design requires operating system mechanisms to support classes as objects and the dynamic loading and execution of objects. These facilities are required to build an object-oriented file system, process mechanism and persistent object implementation scheme. The support that C++ provides in writing these systems is limited because dynamic loading is not supported and classes are compiled into C code. However, we do not consider this restriction in C++ a disadvantage in our implementation because, if they had existed, they might have biased our implementation or forced us to modify the semantics of the language.

The Choices classes that support operating system construction are divided into two portions. The *Germ* is a set of classes that encapsulates the major hardware dependencies of Choices and provides an *idealized* hardware architecture to the rest of the classes in the hierarchy. It provides the *mechanisms* for managing and maintaining the physical resources of the computer. A *Kernel* is a collection of classes that supports the execution of applications and implements resource allocation *policies* using the Germ mechanisms. Individual customized systems consist of instances of Germ classes defined by Choices appropriate for the particular hardware of the system, plus the specifically tailored Kernel classes the system builder desires. Once this instance hierarchy is built, individual applications that run on top of the new Kernel can further augment the Choices class hierarchy with their own classes.

In Choices, both the Germ and Kernel class collections are embedded in a class hierarchy that has class *Object* at its root, see Table 1. Class Object provides virtual functions for the construction and destruction of system instances.

---

[6] For simplicity, Table 2 does not show the disk storage classes.

In the following sections, we will describe some of the classes that constitute Choices.

3. Choices Memory Management

Memory management in Choices is implemented by a class hierarchy derived from the *MemoryRange* class shown in Table 2 and supports virtual memory, the sharing of memory, and memory protection. An instance of a MemoryRange represents a contiguous range of memory addresses, as the name implies. MemoryRange is subclassed for virtual and real memory. *Space* is a subclass of MemoryRange that represents a range of virtual memory that can potentially be addressed by a processor. A Space is a range of virtual addresses; however, an address in the range may be unallocated, reserved and invalid, or reserved and valid (where valid implies that it is mapped to physical memory). Another subclass, *Store*, exists to represent physical addresses in a Choices system. The methods operating on MemoryRanges are specialized for Spaces and Stores and some of them are also shown in Table 2. Many of the methods defined for Spaces are inherited by the subclasses of Space. The methods are summarized below.

3.1. MemoryRange Methods

The constructor for a MemoryRange takes, as a minimum, a base address and length for the range. The parameters of the constructor are augmented in the various subclasses of MemoryRange in order to implement the various specializations of this class.

The most important methods for MemoryRanges are **reserve** and **release**. Reserve records that a sub-range of addresses within the MemoryRange are in use or "reserved". The sub-range to reserve is specified by a starting address and a length argument. Reserve returns an error if any of the addresses are already reserved (and have not yet been released). A method **allocate** (not shown) operates like reserve but takes a single argument, the length to reserve, and reserves a range of unused addresses starting at an address selected by the method rather than by an argument. Functions **isReserved** and **isAvailable** exist to test whether a sub-range is already

Hierarchy of Memory Classes									
Classes	Methods								
MemoryRange	**resrv.**	**release**	**phys.Ad**	**isResrv.**	**isAvl.**	**start**	**end**	**size**	—
↓ Spaces	*resrv.*	*release*	*phys.Ad*	*isResrv.*	*isAvl.*	*start*	*end*	*size*	FixF'lt
↓ ↓ F'ltInSpace	*resrv.*	*release*	*phys.Ad*	*isResrv.*	↓	↓	↓	↓	↓
↓ ↓ ↓ FilledF'ltInSpace	*resrv.*	↓	↓	↓	↓	↓	↓	↓	*FixF'lt*
↓ ↓ PrefetchedSpace	*resrv.*	↓	↓	↓	↓	↓	↓	↓	↓
↓ Stores	*resrv.*	*release*	*phys.Ad*	*isResrv.*	*isAvl.*	*start*	*end*	*size*	—

Hierarchy of SpaceList Classes					
Classes	Methods				
SpaceList	**add**	**remove**	**isIn**	**spaceContainer**	**setEqual**
↓ Domain	*add*	*remove*	↓	↓	↓
↓ Universe	*add*	*remove*	↓	↓	*setEqual*

Table 2: Memory and SpaceList Class Hierarchies

reserved or not.

Other MemoryRange methods include **start**, **end** and **size** that return the base address of, the last address of, and the total number of addresses in the range. A function **isIn** (not shown) returns whether an address lies within the MemoryRange.

3.2. Stores

The Store class represents a range of *physical* or *real* memory. It encapsulates the data structures used by the reserve, allocate, and release methods of Store to manage the allocation of real memory to a process. A Choices operating system may include multiple instances of the Store class; each representing a physical memory with a different property or attribute. For example, a Store instance may represent the local memory private to a processor, a software maintained private cache, the physical memory shared within a multiprocessor, or the physical memory shared between multiprocessor clusters.

3.3. Spaces

The Space class represents a range of *virtual memory* that may be shared and protected. An instance of Space is similar to the notion of a segment, it may, for example, represent a stack, data, or code segment. A Space encapsulates the hardware dependent data structures (for example, the page or segment table entries) that implement the mapping of the virtual addresses in the range to physical memory. The methods reserve, allocate, and release are overloaded to manage the allocation of virtual memory addresses. For example, a request to extend a run-time stack will result in the reservation of the virtual memory addresses for that extension. Spaces augment the set of methods inherited or redefined from MemoryRange with **fixFault**. The fixFault method is used to implement demand paging, prepaging, or segmentation. For example, the Encore Multimax version of fixFault provides demand paging and is invoked by the exception handling mechanism of a processor whenever a memory referencing error, including a page fault or protection violation, is detected by the hardware. The fixFault method validates or maps real memory into the virtual memory addresses in page-size increments. Should the virtual memory addresses correspond to virtual memory that has been paged-out, the method initiates reading the pages from disk into the appropriate real memory. A Space also maintains the access permissions for its virtual address range. Methods are provided to set these permissions. The permissions on a Space apply to all the addresses represented by that Space.

Every Space maps its virtual addresses by referring to a lower-level MemoryRange that is its source of real memory. This MemoryRange may be a Store or, recursively, another Space. A "low-level" Space maps virtual addresses into the real memory of a Store using the methods of the Store to acquire valid real memory addresses. A "high-level" Space maps virtual addresses into the real memory of a Store through an intermediary Space. Recursively, it uses the MemoryRange methods on the Space to acquire valid real memory addresses. The MemoryRange, Space, and Store method **physicalAddress** returns a real memory address corresponding to a virtual address argument. The physicalAddress method for a Space uses the internal virtual to real memory address mappings of the Space to translate valid virtual addresses into corresponding physical addresses. If the virtual address is unmapped (invalid), the Space's method invokes the physicalAddress method of the next lower-level MemoryRange and returns the result of that method. An invocation of a physicalAddress method on a Space can result in a chain of such invocations. This chain continues until either a Space is reached for which a valid mapping for the virtual address is defined or a Store is reached[7]. The physicalAddress method for a Store both reserves real memory and returns its physical address in a similar manner to the allocate method. Thus, eventually, all physicalAddress requests are satisfied.

Spaces allow several processes to share memory. If required, each process may have a different access permission to that memory. The simplest mechanism for sharing a memory is for each process to share a common Space. However, using this approach each process will have the same access permissions to the memory[8]. To overcome this limitation, processes may share memory by having multiple Spaces, each of which implements a different access permission. For example, two processes may each have a Space that maps a given virtual address range into the same real memory locations. Each Space may set up its mapping tables with different access permissions. One process may be granted read/write access to the memory and another may be granted read only access. If a page of memory has to be paged out, both Spaces must invalidate the corresponding virtual memory address range. Spaces also allow a real memory to occupy different virtual address ranges. For example, a given set of memory locations can be shared by mapping it into both a virtual address range of $0 - n$, and a virtual address range of $m - m+n$.

3.4. Space Lists, Domains and Universes

A *SpaceList* provides methods for the aggregation of Spaces and is shown in Table 2. It is specialized into a *Domain* that represents the virtual memory that can be accessed by a user process as it executes an application or a persistent object method. Domains have methods **add** and **remove** that grant or revoke a process' access to particular data and code. In addition, methods are provided to check if a Space is contained within a Domain (**isIn**), and to return the Space within a Domain containing a given address (**spaceContaining**). Another specialization of a SpaceList is the *Universe* that represents the virtual memory and, in particular, the actual hardware translation mechanism of a processor. A Universe is a list of the non-overlapping Spaces that form the virtual memory that is addressable by a processor at any one time. Domains as well as individual Spaces may be added or removed from a Universe. In the Multimax implementation, the Universe maintains the first level page tables and is responsible for keeping the actual memory mapping of the processor consistent with its list of Spaces.

3.4.1. Primitive and Derived Spaces

A process may have rights to access a Space as a *Primitive Space* that contains memory (for example, a process stack, code, or local data), or as a *Derived Space* that contains persistent objects.[9] Primitive Spaces are protected from invalid read, write, or execute access. The contents of a Derived Space cannot be accessed by a process unless it changes its domain of execution to a persistent object that is encapsulated within the Space. This change of domain can only occur by the invocation of a persistent object method. Such an invocation creates a page fault. The Germ then checks that the method invoked is a valid method and that access to that method has been granted to the process. A Domain containing the Primitive Spaces that permit access to the

[7] Usually, there is one Space that is at the lowest-level and this coordinates the reservation and validation of the shared memory.

[8] It is possible to overcome this problem by running some processes in supervisor state and others in user state. This solution is used for certain system functions, but is not a general mechanism.

[9] A Derived Space is created from a Primitive Space by granting processes access rights to the methods of the objects within the Space. In Choices, such objects are called "persistent" because their existence becomes independent of the lifetime of any one process (in particular the one that created it). We emphasize the distinction between a Derived Space and a persistent object. Although a Derived Space can contain persistent objects, the Space itself is a Germ object.

contents of the persistent object is then added to the Universe and the method is executed. The change in Domain may also remove other Domains from the Universe according to protection policies. (However, a discussion of the implementation of protection policies and the naming schemes for persistent objects and their methods is beyond the scope of this paper.)

The next section discusses the Choices concept of a process.

4. The Choices Process Model

Choices was designed to support real-time multiprocessing and parallel computing on large numbers of processors. To facilitate this, Choices supports the concept of a computation that is composed of a potentially large number of lightweight, independent parallel processes. A single one of these processes is represented by an instance of the Choices *Process* class shown in Table 3. An application may use multiple communicating processes to achieve concurrency and parallelism. Each Process represents a small independent sequential computation that can share memory through the memory management mechanisms previously described. Processes exist orthogonally to memory and address management issues. Each Choices Process has a Domain that specifies its associated virtual memory. Usually, the executable code, initialized data, uninitialized data, and stack are represented as separate Spaces within this Domain. The constructor for a process is parameterized by an initial Domain, an initial program counter and stack pointer, and arguments to the process. Methods for Processes alter their Domains, manipulate scheduling parameters, and handle preemption and dispatching.

4.1. Process Context Switching

The state of a process is recorded by storing a stack pointer, a program counter, and a set of register contents within an Process object. A small system stack is maintained by a Process in order to handle hardware preemption. The **dispatch** method reloads a CPU's registers with copies that are stored within the Process. In the Multimax implementation, if the Domain of the Process being dispatched matches the Universe of the processor (the processor may have been executing a process that has a similar domain), no memory context switching is necessary. That is, the Multimax page table entries do not need to be modified and the memory management unit (MMU) does not need to be flushed.

Interrupt and real-time processing require the ability to switch between processes with minimum context switching overhead. Unfortunately an executing process accesses a stack, code, and data represented by the various Spaces contained within its Domain. To accommodate high-performance context switching, processes may lock the memory of a MemoryRange as resident (the locking methods are not discussed in this paper.) Locking memory to be resident within a Space causes the corresponding virtual addresses to be validated and the associated real memory to be locked as resident in physical memory by the corresponding Store. In addition, a process may lock a Space within the CPU's Universe, making the Space and its tables resident and ensuring that the Space's virtual memory mapping is available. A context switch to a process that addresses only resident pages in resident virtual memory creates only the register loading overhead.

Interrupt handlers and real-time processes can be implemented using this high-performance optimization, if desired. Such processes may still be protected from other applications by running the processes in the privileged state of the processor and setting the memory protection of the Spaces in their Domain to exclude access in non-privileged mode. Thus, even though a Space may be locked in the virtual memory of the processor, it can remain protected from unprivileged processes. The Kernel memory of a Choices system is implemented as one such Space.

4.2. Interprocess Communication

Communication between processes can be achieved by means of shared Spaces or by invoking methods[10] on another process's Process. Popular shared memory and message passing communication schemes exist in the system as part of the operating system and are defined within the class hierarchy (not shown in this paper.) Other user defined communication schemes can be built by extending the class hierarchy. An interface compiler for C++ enriches the possible communication schemes. Currently defined are a Path C++ class (named after Path Pascal [5]), and semaphores. Monitors, messages, and simple varieties of guarded commands are currently being designed and implemented.

Protected communication can be achieved by means of shared Spaces containing persistent objects (defined later). The methods of such objects may enforce particular communication protocols upon the processes that use them and the protection provided by the objects methods prevents misuse.

Persistent objects may be *active,* that is, they may have constructors or methods that create "encapsulated" Processes. Such a Process is initialized with a Domain that includes the primitive Spaces of the persistent object. Thus, a process can be directly associated with a persistent object. Active objects can be used to implement name servers and to send asynchronous messages. Several persistent *system* objects augment the shared persistent objects and provide high-performance communication channels between processes and between processes and devices. System objects can support stream-based communications, broadcasts, multicasts, and block I/O.

4.3. ProcessContainers and Scheduling

Scheduling and dispatching issues in Choices are handled by instances of the *ProcessContainer* class shown in Table 3. A ProcessContainer, as the name implies, is a container of

Hierarchy of ProcessContainer Classes					
Classes	Methods				
ProcessContainer	**add**	**remove**	**isEmpty**	—	—
↓ SingleProcessHolder	*add*	*remove*	*isEmpty*	—	—
↓ ↓ LockedSn'glPrc'sHl'dr	*add*	*remove*	↓	—	—
↓ ↓ CPU	*add*	*remove*	↓	**disableIntrpt.**	**enableIntrpt.**
↓ FIFOScheduler	*add*	*remove*	*isEmpty*	—	—
↓ ↓ RRScheduler	↓	*remove*	↓	—	—

Hierarchy of Process Class					
Classes	Methods				
Process	**dispatch**	**domain**	**ch'gDomain**	**sch'Info**	**setSch'Info**

Table 3: Process and ProcessContainer Class Hierarchy

[10] Such a method is similar in intent to the UNIX signal.

Processes. It is the basic entity of scheduling and dispatching in any Choices system. All scheduling issues involve moving Processes between ProcessContainers.

The queueing discipline that a ProcessContainer uses depends entirely on the subclass of the ProcessContainer hierarchy being used. The top level ProcessContainer class itself is abstract and only defines the operations **add** (for inserting Processes into the container), **remove** (for removing the next available Process from the container), and **isEmpty** (for testing whether the container is empty or not.) Subclasses redefine these methods, for example, to add and remove Processes in FIFO, LIFO, or priority order (not shown.)

ProcessContainers implement the "traditional" run/ready/blocked queue models of operating systems. ProcessContainers are also used to store Processes that await an event or are blocked on a semaphore operation. A special subclass of ProcessContainer, *CPU*, has an add method that "stores" and executes a Process on a processor. The CPU remove method is used to model preemption. The CPU is conceptually a special type of ProcessContainer that invokes the dispatch method on a Process that is placed into it. Multiprocessing fits naturally into the Choices model of scheduling since a Choices system can consist of more than one CPU object. Persistent objects are discussed further in the next section.

5. Persistent Objects

Choices is designed with the objective of placing many operating system and subsystem components in a protected Space rather than in a kernel as is done in traditional systems. This is done to reduce the interdependences among operating system components and to increase the coherence of the components themselves. Such components are implemented as Choices *persistent objects.* That is, instances of classes that reside in memory for periods that exceed the execution of a particular process and that may be shared between multiple processes. Persistent objects may be mapped into the virtual memory of several processors at the same time using the Space shared memory implementation. In a sense, the Germ and Kernel of a Choices system are collections of persistent objects that are always resident and accessible (in a controlled manner) in the address space of every processor.

A full description of the protection scheme used in Choices is beyond the scope of this paper. However, we must introduce enough of the scheme here in order to describe access to (and the invocation of methods on) a persistent object. Each process executes within a protection domain that dictates what the process may access. The protection domain of a process is dynamic and may change by adding or removing Spaces. Initially, the protection domain depends upon the protection of the executable file that the process is created from and the protection domain of the parent process. A process that executes a method of a persistent object enters a new protection domain that depends upon the protection of the Derived Spaces containing the object and the protection domain of the process. When the process returns from the method invocation, its previous protection domain is restored. The scheme is implemented using the memory management classes introduced in §3.

For example, policy modules of the operating system that traditionally are part of the kernel, may be implemented as persistent objects. A process executing one of the methods within these persistent objects may require access to Germ data structures (typically also requiring some sort of "supervisor" execution privileges). This is possible by having the process enter the protection domain of the persistent object (which would include the change to the supervisor execution level) via the *gate* mechanism. The gate mechanism implements the controlled entry of the process into the new protection domain. When the invoked persistent object method returns, the process' protection domain is restored to what it was before the method call.

Processes access persistent objects using an *object descriptor* and a method. A process must obtain the object descriptor before use. Object descriptors are provided from user or system name servers.

Name servers are themselves persistent objects. Choices includes "standard name servers" that are in the Kernel and may be accessed by every process. These name servers provide basic facilities like the standard file system and intertask communication. Other user defined name servers must be accessed through the standard name server utilities.

On request, the name server grants the process access to the object and returns an object descriptor for the requested object. The grant operation is implemented in the Germ and checks Kernel protection policy to determine if the name server/process grant operation is valid. The name server must have appropriate access rights to the persistent object. If the operation is valid, the Germ adds the Space of the persistent object to the Domain of the Process, and returns the Space address and gate information to the name server. The name server packages an object descriptor which includes the persistent object, Space and gate information and returns.

An operation on a persistent object is invoked through a *gated request*. The Germ ensures that the object descriptor and method used by the process' gated request correspond to the valid persistent object address and method entry point within the Space. The Domain of the Process is changed to reflect the protection domain requirements of the Space.

In hardware architectures with limited virtual memory, the gated method of invoking a persistent object allows many different Spaces to share the same virtual memory address range. The Space and the persistent objects it contains can be mapped into and out of the same address range on demand[11]. In such implementations, the Domain will contain each Space, but only one of the Spaces will be present in the Universe at any one time.

6. Exception Handling in Choices

Exceptions in Choices are managed by the *Exception* class and its various subclasses shown in Table 4. The parent class of Exception defines the method, **raise**, to manage or correct the exception condition. Upon an exception condition, the Choices Germ manages the task of converting the machine dependent details of exception processing into an invocation of the raise method for the Exception object managing the exception.

Two subclasses of Exception of interest are *Trap* and *Interrupt*. The Trap class provides Choices with a mechanism for handling traps that a process may generate as a direct result of its execution. This includes machine traps (for example, divide-by-zero and illegal instruction), virtual memory access and protection errors (for example, page faults of various types), and explicit program traps (for example, a "system call".)

The basic function of a Trap handler is, if possible, to service the exception condition within the context of the faulting Process, or otherwise to terminate the execution of the faulting Process.

Interrupts occur asynchronously and, in general, have nothing to do with the currently executing process. In Choices, the await method of an Interrupt can be used to specify a Process that must be executed when the Interrupt is raised. (Interrupts *must* be awaited if they are not to be missed). The raise method of the Interrupt class saves the context of the interrupted process and

[11] In many hardware architectures, a persistent object must be relocated by a link editor to allow it to execute within a specific address range. This implies that once it is activated, it cannot be moved to a new address range.

Hierarchy of Exception Classes		
Classes	Methods	
Exception	**raise**	—
↓ HardwareException	↓	—
↓ ↓ AbortTrap	*raise*	**fixdPgFlt.**
↓ ↓ SVCTrap	*raise*	—
↓ ↓ IllegalInstructionTrap	*raise*	—
↓ ↓ UndefinedInstructionTrap	*raise*	—
↓ ↓ DivideByZeroTrap	*raise*	—
↓ ↓ InterruptException	*raise*	**await**
↓ ↓ TimeSliceInterrupt	*raise*	**clockTick**
↓ SoftwareException	*raise*	**handler**
↓ ↓ TwoProcessContainerException	↓	*handler*
↓ ↓ OneProcessContainerException	↓	*handler*
↓ ↓ GarbageException	↓	—

Table 4: Exception Class Hierarchy

resumes the Process awaiting the occurrence of the interrupt. The Choices Germ has no requirement that all interrupts be handled by the class Interrupt. A Choices kernel implementer can choose to have any type of Exception object handle an interrupt. In future work, various user-oriented exception schemes will be implemented as classes and by the interface compiler. Examples of such schemes can be found in [6].

7. Experiences Using C++ as a Systems Programming Language

It is not often that it is possible to conduct operating systems research using a new language. Since the C++ compiler[12] generates C and we were consequently able to port it to the Encore Multimax in just a few hours, we were able to do all our development in C++. No additional vendor support for C++ was required. In addition, we were able to exploit the Encore C compiler for the Multimax which produces highly optimized code. Because we are using the same C compiler for measuring performance of algorithms on the Multimax under UNIX and under Choices, our measurements more accurately reflect the differences in the operating systems rather than in their compilers.

Overall, the implementation of Choices has benefited greatly from the availability of C++, its performance, and its compiler. Run-time overhead is negligible, even in the case of *virtual* member functions (class methods). Because of the small overhead, the authors feel that virtual member functions could have been defined as the default in the language since their semantics more closely model the desired behavior of method (member function) redefinition in class hierarchies.

We have encountered some minor problems and issues that have not been a major cause for concern but, from our (perhaps naive) perspective, could be addressed and improved in future

[12] The AT&T C++ Translator.

releases of C++.

The order in which the constructors for *statically allocated* objects are invoked should be under the control of the programmer (or at least defined). For example, during the boot of Choices we have preferred to initialize the Console object before other objects in order to permit the constructors of other system objects to print diagnostic messages.

We believe that classes should be "first-class" objects. That is, a class should be an entity on which a method can be invoked. This would provide a number of advantages. The existence of *class* data and methods would be more consistent with the way in which *instance* data and methods are modeled.[13] For example, **new** and **delete** could be class methods. This would allow easier (and cleaner) customized memory allocation strategies to be implemented on a per-class basis.

Next, class objects would simplify the implementation of dynamic method binding. For example, virtual function tables (which currently do not have unique instances) could be stored as class data. Each instance of a class could include a reference to its class object, so that a change to the class object's virtual function table could be used to change the binding of a method of every instance of the class.

Last, class objects would allow a solution to the static constructor ordering problem. Constructor dependency information could be included in the class data, and could then be topologically sorted at link-time to determine a correct calling order.

8. Summary

A Choices Kernel currently runs on a 10 processor Encore Multimax that supports the MemoryRange model of memory management as well as the Process and Exception concepts.

Of particular concern during the development of the system is whether or not the class hierarchical approach can support the construction of *entire* operating systems. C++ was chosen as an implementation language because it supports class hierarchies and inheritance while

Preliminary Choices Performance Data Encore Multimax 32032 (0.75 MIP)		
Operation	Encore 4.2 BSD Unix	Choices
System Call Overhead	173μsec	39μsec
Process Creation	26.3msecs	3.8msecs
Context Switch	—	536μsecs
Shared Memory Example[14]	0.032secs	0.022secs

Table 5: Performance Data

[13] Currently, class *data* are obtained using the keyword "static" in the class definition (there are no class *methods*.)

[14] The example creates four processes on independent processors, three sum a ten column array and the fourth sums the three resulting sums. The Multimax multitasking library package was used under UMAX.

imposing negligible performance overhead at run–time. A software monitor is being used to evaluate the performance of Choices on an Encore Multimax with DPC processors. Although it is difficult to provide a meaningful performance measurement of an operating system, we have obtained results that are encouraging and these are shown in Table 5. System call overhead (including a trap and change to supervisor state) compares favorably with UNIX and is only about four times the overhead of a normal procedure call. The process creation time includes creation of new virtual memory "spaces" for the process. Further tuning will improve these figures.

Current effort is devoted towards improvement and further implementation of communication and persistent object support. Future plans include an object-oriented file system, an advanced interface compiler, and tools for configuring Choices systems. Once Choices is stable, the code will be placed in the public domain to promote research into customized operating systems.

References

1. Accetta, Mike, et. al. "Mach: A New Kernel Foundation for UNIX Development." USENIX Conference Proceedings, June 1986, pages 93–111.
2. Allchin, J. E. and M. S. McKendry. "Support for Actions and Objects in Clouds: Status Report." Georgia Institute of Technology Technical Report GIT–ICS–83/1, Atlanta, Georgia, January 1983.
3. Brownbridge, D. R., L. F. Marshall, and B. Randell. "The Newcastle Connection, or UNIXes of the World Unite!" Software – Practice and Experience, 1982, pages 1147–1162.
4. Campbell, Roy H., Gary M. Johnston, and Vincent F. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)." Operating Systems Review, Vol. 21, No. 3, July 1987, pages 9–17.
5. Campbell, Roy H. and T. J. Miller. "A Path Pascal Language." Department of Computer Science Technical Report UIUCDCS–R–78–919, University of Illinois at Urbana–Champaign, Urbana, Illinois, April 1978.
6. Campbell Roy H. and B. Randell, "Error Recovery in Asynchronous Systems." IEEE Transactions on Software Engineering, Vol. SE–12, No. 8, August 1986, pages 811–826.
7. Cheriton, David R. and Willy Zwaenepoel. "Distributed Process Groups in the V Kernel." ACM Transactions on Computer Systems, May 1985, pages 77–107.
8. Encore. "Encore Multimax Technical Summary." Encore Computing Corporation, 1986.
9. Li, Kai and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems." Proceedings of the Fifth Annaul ACM Syposium on Principles of Distributed Computing, August 1986, pages 229–239.
10. Lu, P. M. "A System for Resources Sharing in a Distributed Environment--RIDE." Proceedings of the IEEE Computer Society Third COMPSAC, IEEE Press, New York, 1979.
11. Popek, G., B. Walker, et. al. "LOCUS: A Network Transparent High Reliability Distributed System." Proceedings of the Eighth Symposium on Operating Systems Principles, printed as Operating Systems Review, Vol. 15, No. 5, December 1981.
12. Rashid, Richard F. and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel." Proceedings of the Eighth Symposium on Operating Systems Principles, printed as Operating Systems Review, Vol. 15, No. 5, December 1981, pages 64–75.
13. Rifkin, Andrew P., et. al. "RFS Architectural Overview." USENIX Conference Proceedings, Atlanta, Georgia, 1986.
14. Robison, Arch. D. "A Functional Programming Interpreter." M.S. Thesis and Department of Computer Science Technical Report UIUCDCS–R–87–1714, University of Illinois at Urbana–Champaign, Urbana, Illinois, March 1987.
15. Sequent. "Balance 8000 Guide to Parallel Programming." Sequent Computer Systems, July 1985.
16. Snyder, Lawrence. "Formal Models of Capability–Based Protection Systems." IEEE Transactions on Computers, Vol. C–30, No. 3, March 1981.

17. Swinehart, Daniel, Polle Zellweger, Richard Beach, and Robert Hagmann. "A Structural View of the CEDAR Programming Environment." Transactions on Programming Languages and Systems, October 1986, Vol. 8, No. 4, pages 419–490.

18. Tanenbaum, Andrew S. and Sape J. Mullender. "An Overview of the Amoeba Distributed Operating System." Operating Systems Review, July 1981, pages 51–64.

19. Walsh, Dan, et. al. "Overview of the Sun Network File System." USENIX Conference Proceedings, January 1985, pages 117–124.

20. Wittie, L. D. and A. Van Tilborg. "MICROS - A Distributed Operating System for MICRONET - A Reconfigurable Network Computer" in *Tutorial: Microcomputer Networks*, H. A. Freeman and K. J. Thurber, eds., IEEE Press, 1981, pages 138–147.

21. Wulf, William A., et. al. "HYDRA: The Kernel of a Multiprocessor Operating System." Communications of the ACM, June 1974, pages 337–345.

BIBLIOGRAPHIC DATA SHEET

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1388	2.	3. Recipient's Accession No.

4. Title and Subtitle
The Design of a Multiprocessor Operating System

5. Report Date
December 1987

6.

7. Author(s)
Roy Campbell, Vincent Russo, Gary Johnston

8. Performing Organization Rept. No. R-87-1388

9. Performing Organization Name and Address
Department of Computer Science
1304 W. Springfield
240 Digital Computer Lab
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NASA NSG 1471
AT&T MetroNet

12. Sponsoring Organization Name and Address
NASA Langley Research Center // AT&T Information Systems
Hampton, VA 23665 // Middletown, NJ 07738

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Evolving applications and hardware are creating new requirements for operating systems. Real-time systems, parallel processing, and new programming paradigms require large adaptive maintenance efforts to modernize existing operating systems. An alternative to such adaptive maintenance is to seek new operating system designs that exploit modern software engineering techniques and methodologies to build appropriately structured modular software. This paper describes an approach to constructing operating systems based on a class hierarchy and object-oriented design, and discusses the benefits and difficulties of realizing this design using C++.

17. Key Words and Document Analysis. 17a. Descriptors
multiprocessor operating systems
object-oriented operating systems
customizable operating systems
embedded systems

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 21
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71